SUBMITTED TO: 11TH SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS

OPTIMISTIC MESSAGE LOGGING FOR INDEPENDENT CHECKPOINTING IN MESSAGE-PASSING SYSTEMS [1]

Yi-Min Wang and W. Kent Fuchs

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
1101 W. Springfield Ave.
University of Illinois
Urbana, IL 61801

Primary contact: Yi-Min Wang

E-mail: ymwang@crhc.uiuc.edu
Phone: (217) 244-7161
FAX: (217) 244-5686

ABSTRACT

Message-passing systems with communication protocol transparent to the applications require message logging to ensure the consistency between checkpoints. This paper describes a periodic independent checkpointing scheme with optimistic logging to reduce the performance degradation during normal execution while keeping the recovery cost acceptable. Both time and space overhead for message logging can be reduced by detecting messages that need not be logged. A new checkpoint space reclamation algorithm is presented to reclaim all checkpoints which are not useful for any possible future recovery. Communication-trace-driven simulation for several hypercube programs is used to evaluate the techniques.

Key words: independent checkpointing, recovery lines, optimistic message logging, message-passing systems.

[1] *Acknowledgement:* This research was supported in part by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283.

1 Introduction

Numerous checkpointing and rollback recovery techniques have been proposed in the literature for parallel systems. In terms of checkpointing techniques, they can be classified into two basic categories. *Coordinated checkpointing* schemes synchronize computation with checkpointing by coordinating processes during a checkpointing session in order to maintain a consistent recovery line [1, 2, 3, 4]. Each process only keeps the most recent successful checkpoint. Rollback propagation is avoided at the cost of potentially significant performance degradation during normal execution. *Independent checkpointing* schemes replace the above synchronization by dependency tracking and possibly message logging [5, 6, 7, 8] in order to preserve process autonomy. Possible rollback propagation in case of a fault is handled by searching for a consistent system state based on the dependency information. Lower run-time overhead during normal execution is achieved by maintaining multiple checkpoints and allowing slower recovery.

In terms of message logging techniques, there are also two primary categories: pessimistic and optimistic. Pessimistic logging protocols avoid rollback propagation by logging each message synchronously [9, 10, 11], i.e., the receiver is blocked until the message is logged on stable storage. Fast local recovery of a failed process is therefore achieved at the expense of greater run-time overhead or specialized hardware. Optimistic logging protocols log messages asynchronously after the receipt [6, 8, 12]. Several messages can be grouped together and written to the stable storage in a single operation to reduce the logging overhead. Rollback propagation can occur if some received messages are not yet logged when an error is detected.

This paper considers the independent checkpointing scheme for possibly nondeterministic execution. We address the problems of application-level checkpointing for systems in which applications access the message-passing capabilities through system calls. Since the low-level communication protocol is transparent to the applications, optimistic message logging pro-

tocol is employed to ensure the consistency of checkpoints without incurring large run-time overhead. Because the recovery line is unknown during normal execution for independent checkpointing, the problem of recording the state of the channels through message logging is more involved than the case of coordinated checkpointing. An algorithm is described in this paper for effectively reducing the number of messages required to be logged. A new checkpoint space reclamation algorithm is also presented to further reduce the space overhead for maintaining multiple checkpoints.

The outline of the paper is as follows. Section 2 describes the system model and the checkpointing and rollback recovery protocol; Section 3 discusses the consistency between checkpoints; modification to the protocol for application-level checkpointing is described in Section 4; Section 5 presents techniques for reducing the number of logged messages and the checkpoint space overhead; and experimental results are described in Section 6.

2 System Models

The system considered in this paper consists of a number of concurrent processes for which all process communication is through message passing. Processes are assumed to run on fail-stop processors [13] and each processor is considered as an individual *recovery unit* [6]. Since studies [14] have shown that the support for nondeterministic processes is important for practical applications, we do not assume deterministic execution. Consequently, if the sender of a message is rolled back, the corresponding message log will be invalid during reexecution, which means that the receiver also has to be rolled back. We do not address the problem of concurrent rollbacks due to multiple failures [7].

We consider systems which provide a set of system calls for the applications to access the message-passing capability, for example, as in the Intel iPSC/2 hypercube. Such systems

hide the detailed communication protocol from the user and usually allow easier programming. However, they impose the following constraints for implementing application-level checkpointing:

1. The state of the communication protocol is invisible to the applications. Information such as "a certain message has arrived at its destination" or "a message has been lost" is typically not available to the applications [15].

2. Although messages are normally first-in-first-out through the communication channel, they are not necessarily received by the application processes in that order. Each message is usually assigned a type and the system calls for receiving messages can specify the message types [16]. Out-of-order delivery of messages is therefore possible at the application level.

During normal execution, the state of each processor is occasionally saved as a *checkpoint* on stable storage. Let CP_{ik} denote the kth checkpoint of processor p_i with $k \geq 0$ and $0 \leq i \leq N-1$, where N is the number of processors. A *checkpoint interval* is defined to be the time between two consecutive checkpoints on the same processor and the interval between CP_{ik} and $CP_{i(k+1)}$ is called the kth checkpoint interval. Each message is tagged with the current checkpoint interval number and the processor number of the sender. Each processor takes its checkpoint independently and includes in each checkpoint the *communication information*, or *input information* [7] containing pairs of processor number and checkpoint interval number, (j, m), if at least one message from the mth checkpoint interval of processor p_j has been received during the previous checkpoint interval.

A centralized *checkpoint space reclamation algorithm* can be invoked by any processor occasionally to reduce the space overhead. First, the communication information of all the existing checkpoints is collected to construct the *checkpoint graph*. The *rollback propagation*

algorithm [5] (shown in Fig. 1) is executed on the checkpoint graph to determine the *global recovery line*. All the checkpoints taken before the global recovery line then become *obsolete* and their space can therefore be reclaimed. An example showing the above procedure is given in Fig. 2(a)-(c) (ignore symbols "V" and dashed boxes for now).

```
/* CP stands for checkpoint. Initially, all the CPs are unmarked */
include the latest CP of each processor in the root set;
mark all CPs strictly reachable from any CP in the root set;
while (at least one CP in the root set is marked) {
    replace each marked CP in the root set by the latest unmarked CP on the
    same processor;
    mark all CPs strictly reachable from any CP in the root set;
}
the root set is the recovery line.
```

Figure 1: The Rollback Propagation Algorithm.

When processor p_i detects an error, it sends out a *rollback-initiating* message [7] to every other processor to request the up-to-date communication information. Each surviving processor takes a *virtual checkpoint* upon receiving the *rollback-initiating* message (Fig. 2 (a) and (b)). After receiving the responses, p_i constructs the *extended checkpoint graph* [5] and executes the rollback propagation algorithm to determine the *local recovery line* (Fig. 2(d)). A *rollback-request* message containing the local recovery line is then sent to each processor and requests the involved processors to rollback and restart.

3 Consistency of Checkpoints

There are two situations concerning the consistency between two checkpoints. In Fig. 3(a),

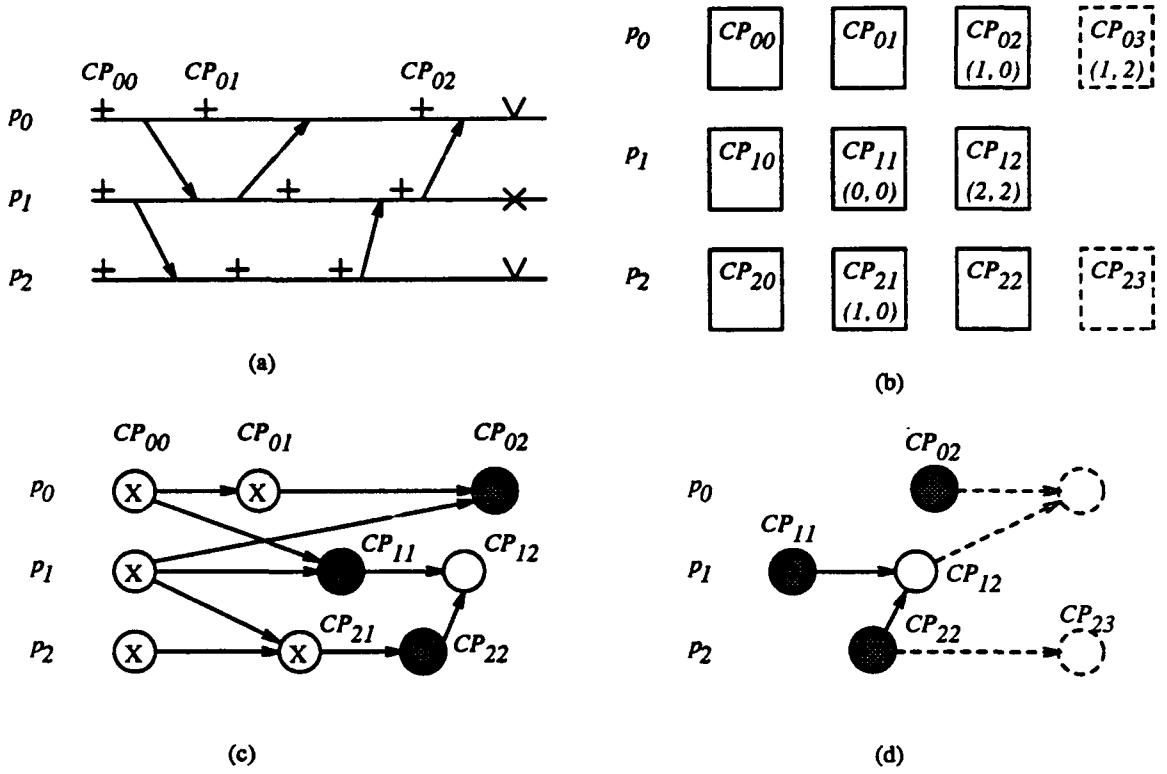

Figure 2: (a) The communication pattern ("+" represents a checkpoint, "V" stands for virtual checkpoint and "X" indicates a detected error) (b) the communication information (c) the checkpoint graph for space reclamation (checkpoints marked "X" are obsolete) (d) the extended checkpoint graph for rollback recovery (CP_{02}, CP_{12} and CP_{23} forms the local recovery line).

if p_i and p_j restart from the checkpoints CP_{ik} and CP_{jm} respectively, the message m will be recorded as "received but not yet sent". Without the assumption of deterministic execution, message m is not guaranteed to be re-sent during reexecution. CP_{ik} and CP_{jm} are thus inconsistent and cannot exist on the same recovery line. By the construction of the checkpoint graph, the inconsistency between CP_{ik} and CP_{jm} is indicated by the edge connecting them. In general, two checkpoints will be inconsistent if there exists a directed path between them on the checkpoint graph.

Fig. 3(b) illustrates the second situation. The message m is recorded as "sent but not yet

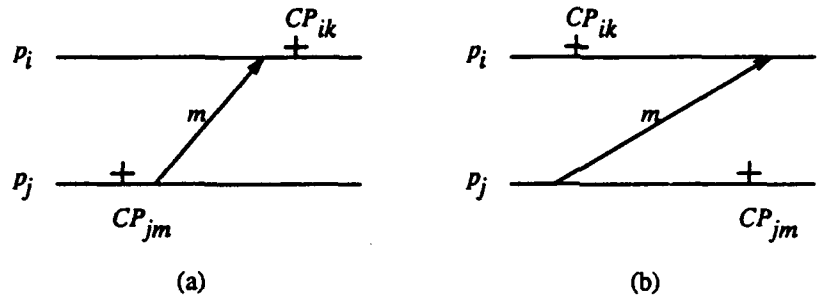

Figure 3: (a) message received but not yet sent; (b) message sent but not yet received.

received" according to the system state containing CP_{ik} and CP_{jm}. Bhargava and Lian [7] defined consistency in such a way that CP_{ik} and CP_{jm} are considered consistent. Koo and Toueg [3] explained that the situation of restarting from CP_{ik} and CP_{jm} is indistinguishable from the situation that message m is lost during normal execution. Therefore, an end-to-end transmission protocol which can guarantee the redelivery of lost messages can also guarantee the resending of message m during reexecution. Under such an assumption, CP_{ik} and CP_{jm} are consistent.

However, as mentioned in Section 2, such a transmission protocol state is not available to the applications in the system model considered in this paper. For example, in an iPSC/2 hypercube, a sending process can proceed once it is informed that the communication hardware will take care of the reliable delivery of the message. Therefore, the process is unable to know when the message arrives at the receiver. Without such information, CP_{ik} and CP_{jm} in Fig. 3(b) can not be considered as consistent unless message m is somehow recorded.

By defining the *state of the channels* to be the set of messages sent but not yet received, Chandy and Lamport [2] proved that checkpoints like CP_{ik} and CP_{jm} in Fig. 3(b) can become consistent if the corresponding state of the channels is also recorded. This concept has typically been applied to coordinated checkpointing techniques with message logging [4, 17]. In these schemes, because only the checkpoints with the same checkpoint number can form

a recovery line, messages comprising the state of the channels can be easily identified. For example, in Fig. 4, message m_0 belongs to the channel state corresponding to the recovery line containing CP_{i1} and CP_{j1} and needs to be logged. Message m_1, however, does not belong to the channel state of any recovery line and needs not be logged. The situation becomes more complicated when we consider the independent checkpointing scheme because the recovery line is known only at the time of recovery. Message m_1 can potentially become part of the channel state if CP_{i1} and CP_{j2} forms a recovery line. We will first discuss how to modify the checkpointing and rollback recovery protocol in the next section and then describe how we can still identify some messages which can not belong to any channel state.

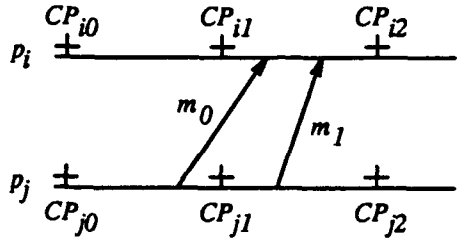

Figure 4: Both messages m_0 and m_1 can become part of the channel state in an independent checkpointing scheme.

4 Modified Checkpointing and Recovery Protocol

4.1 Valid Checkpoints

First we discuss a possible erroneous situation where the checkpoints are not added to the checkpoint graph in the correct order because of the unpredictable message transmission delay during the collection process. Note that when we reclaim the space for "obsolete" checkpoints, an implicit assumption is that the global recovery line will always move forward

in time. This is true only if checkpoints taken earlier are always added to the checkpoint graph earlier. However, the situation in Fig. 5 might occur. Referring to the same example in Fig. 2, checkpoint CP_{22} was taken earlier than CP_{12}. Suppose the communication information (Fig. 2(b) for CP_{22} is not collected by the time the communication information for CP_{12} is collected (Fig. 5). If we simply include CP_{12} in the checkpoint graph and leave out all the edges connected to CP_{22}, the rollback propagation algorithm will determine that $\{CP_{02}, CP_{12}, CP_{21}\}$ forms the global recovery line and all the earlier checkpoints can be reclaimed. However, this results in an erroneous situation because CP_{12} and CP_{21} are in fact inconsistent. It becomes clear that a checkpoint should only be included in the checkpoint graph when all of its incoming edges are included.

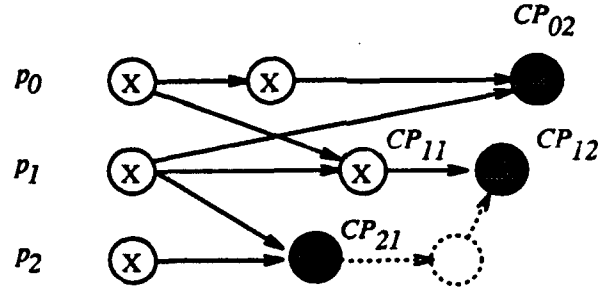

Figure 5: Erroneous global recovery line containing inconsistent checkpoints CP_{12} and CP_{21} due to the missing checkpoint CP_{22}.

A similar situation might occur in another scenario when message logging is used to record the channel state. Suppose one more message m is added to the communication pattern in Fig. 2(a), as shown in Fig 6. The result of executing the checkpoint space reclamation algorithm remains the same as in Fig. 2(c). However, if message m is not yet logged when an error is detected on processor p_1, the rollback of p_1 will request the resending of message m from p_0, resulting in the rollback of p_0 to CP_{01}. This is equivalent to adding an edge from CP_{12} to CP_{02} on the extended checkpoint graph (Fig 6(b)). Because CP_{01} was reclaimed,

the recovery will fail. This erroneous situation arises because CP_{02} should not have been included for space reclamation. The fact that message m was not yet logged implied a potential edge from CP_{12} to CP_{02} which was not included for space reclamation.

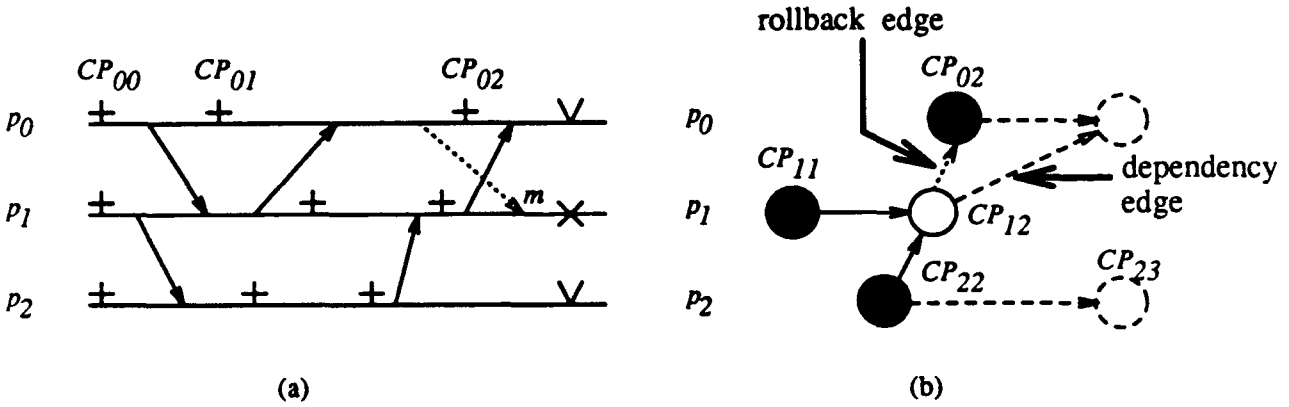

Figure 6: (a) The communication pattern with an extra message m (b) the rollback edge implied by m causes the recovery to fail.

We will refer to the normal edges representing the communications as *dependency edges* as opposed to the *rollback edges* shown in Fig 6(b). We define a checkpoint to be *valid* if it can be included in the checkpoint graph without resulting in the above erroneous situations. A checkpoint is then valid if the following two conditions hold:

1. the source checkpoints of all incoming dependency edges are valid and
2. all the messages sent before the checkpoint have been logged (or need not be logged as explained later).

4.2 Modifying the Protocol

Based on the notion of valid checkpoints, the checkpointing and rollback recovery protocol described in Section 2 can then be modified as follows.

1. Each message is assigned a *sequence number* which will be included in the communication information of the receiver. The sequence number is incremented every time a message is sent.

2. Each processor logs the messages it has received during the previous checkpoint interval at the time it takes a new checkpoint. Pairs of sequence number and destination processor number, (q, j), are also recorded for each message (with sequence number q) it has sent to processor p_j during the previous checkpoint interval.

3. Before invoking the rollback propagation algorithm for space reclamation or for recovery, the set of valid checkpoints are first determined by checking the messages sent against the messages received and logged according to the global communication information collected. If any message is sent from the kth checkpoint interval of processor p_i to processor p_j and is not yet logged by p_j, all checkpoints CP_{im} with $m > k$ are invalid and excluded from the checkpoint graph.

5 Reducing the Space Overhead

5.1 Reducing the Number of Message Logs

As mentioned in Section 3, the difficulty of recording the channel state by message logging with an independent checkpointing scheme lies in the fact that the recovery line is unknown during normal execution. If all messages have to be logged, both time and space overhead might be prohibitively large. The following theorem gives a sufficient condition for identifying messages that will never become part of any channel state, referred to as *non-state messages*, and thus need not be logged.

THEOREM 1 *If there exists a path from CP_{ik} to CP_{jm}, then all the messages sent from the $(m-1)$th checkpoint interval of processor p_j and received by processor p_i at the kth checkpoint interval are non-state messages.*

Proof. Suppose the channel state corresponding to a recovery line R contains any of such messages. By definition, R must contain CP_{jx} with $x \geq m$ and CP_{iy} with $y \leq k$. By the construction of the checkpoint graph, there must exist a path from CP_{iy}, $y \leq k$, to CP_{ik} because any checkpoint must have dependency on the previous checkpoints of the same processor. Similarly, there is a path from CP_{jm} to CP_{jx}, $x \geq m$. If there also exists a path from CP_{ik} to CP_{jm}, the path from CP_{iy} to CP_{ik} to CP_{jm} to CP_{jx} implies CP_{iy} and CP_{jx} are inconsistent, contradicting the fact that CP_{iy} and CP_{jx} are on the same recovery line R. Therefore, all the messages satisfying the statement of the theorem are non-state messages.

□

From another point of view, the potential rollback edge resulting from any of the above non-state messages is from CP_{ik} to CP_{jm}. Since the inconsistency between CP_{ik} and CP_{jm} is already implied by the dependency path between them, adding the rollback edge will not affect the recovery line. Theorem 1 is useful in two ways. First, when the checkpoint space reclamation algorithm is invoked, the space for message logs corresponding to the non-state messages can be reclaimed. Second, notice that our optimistic logging protocol logs received messages only at the end of the checkpoint interval. If a processor can detect the path information as described in Theorem 1 during the current checkpoint interval, it does not have to log all the received messages. In order to reduce the overhead of dependency tracking for detecting paths, we implement an algorithm which only detects edges, instead of paths, by monitoring the message exchange during current checkpoint interval. The algorithm will be described in Section 6 along with other implementation issues.

5.2 A Checkpoint Space Reclamation Algorithm

In addition to the message logs, the checkpoints constitute another space overhead. Traditional checkpoint space reclamation algorithms only reclaim obsolete checkpoints, i.e., checkpoints older than the global recovery line. All *non-obsolete* checkpoints are assumed to be possibly useful for future recovery and therefore need to be kept on stable storage. When the domino effect [18] occurs, a large number of these non-obsolete checkpoints results in large space overhead. We define a *discardable checkpoint* as a checkpoint that will never belong to any future recovery line, either global or local. Obsolete checkpoints are clearly discardable. The following two examples will show that some of the non-obsolete checkpoints are also discardable.

Fig. 7(a) is a typical example for illustrating domino effect. The global recovery line stays at the set of starting checkpoints and is unable to move forward. The edge from CP_{02} to CP_{12} and the one from CP_{11} to CP_{02} implies that CP_{02} is inconsistent with any checkpoint of processor p_1. Since a recovery line must contain one checkpoint from each processor, CP_{02} will never belong to any recovery line and is therefore discardable. [2] CP_{11} and CP_{01} are discardable by similar arguments.

Fig. 7(b) shows a situation where a processor does not receive any message from any other processor during a checkpoint interval. CP_{12} has only one incoming edge from CP_{11}. All the checkpoints connected to CP_{12} through a path must also connected to CP_{11}. In other words, all the checkpoints which are able to form a recovery line with CP_{11} must also be able to form a "better" recovery line with CP_{12}. Therefore, CP_{11} can be discarded.

By using the model of *maximum-sized antichains* on a *partially ordered set* [20, 21, 22]

[2] The situation becomes more complicated when some checkpoints can be removed from the checkpoint graph due to rollback recovery. We have been able to show [19] that all the results in this section are also valid for such a situation.

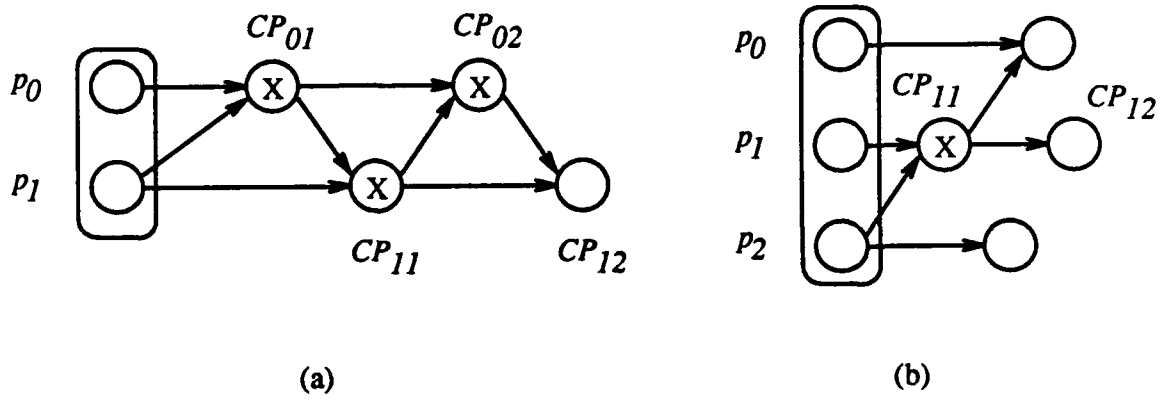

Figure 7: (a) CP_{01}, CP_{11} and CP_{02} are discardable (b) CP_{11} is discardable because CP_{12} has only one incoming edge.

to predict the possible future recovery lines, the following necessary and sufficient condition for a checkpoint to be non-discardable has been proved. Interested readers are referred to [19] for detailed derivations.

THEOREM 2 *Let N be the number of processors and $\hat{G}$ be the supergraph of a checkpoint graph G by adding one checkpoint n_i to each processor p_i in the way shown in Fig. 8(b). A checkpoint in G is non-discardable if and only if it belongs to the union of the recovery lines of $\hat{G} - n_i$, $0 \leq i \leq N-1$.*

Theorem 2 also gives the algorithm for finding the set of non-discardable checkpoints: execute the rollback propagation algorithm on each $\hat{G} - n_i$ and then take the union of the recovery lines. The complexity of the rollback propagation algorithm as described in Fig. 1 is linear in the number of edges $|E|$ because each edge can be removed once it is visited. Our Predictive Checkpoint Space Reclamation (PCSR) algorithm is then of complexity $\mathbf{O}(N|E|)$. Since every non-discardable checkpoint determined by the PCSR algorithm belongs to the recovery line of $\hat{G} - n_i$ for some i, which is a possible future graph of G, the number of checkpoints reclaimed by our algorithm is maximum. An example for running the PCSR

algorithm is shown in Fig. 8(c)-(f).

Another contribution of Theorem 2 is that it implies the number of non-discardable checkpoints is bounded by N^2 because each of the N recovery lines consists of N checkpoints. By further exploiting the relationship among these N recovery lines, an upper bound, $N(N+1)/2$, has been derived [19]. The space overhead for maintaining multiple checkpoints is therefore bounded even when the domino effect persists during program execution.

6 Experimental Results

Four numerical and CAD programs written for the Intel iPSC/2 hypercube are used to evaluate the proposed scheme. The periodic checkpointing routine is implemented as the interrupt service routine for UNIX *alarm(T)* system call, where T is the checkpoint interval in seconds. Each node processor sets the alarm at the very beginning of the node program and the checkpointing routine independently. A concurrent checkpointing algorithm as described by Li et. al [23] is assumed so that the program thread is interrupted for a small, fixed amount of time (0.1 seconds) for taking each checkpoint, after which the checkpointing thread executes concurrently with the program thread to finish the checkpointing. Communication traces are collected by intercepting the "send" and "receive" system calls. Communication-trace-driven simulation is then performed to obtain the results.

We refer to the above checkpointing scheme as "loosely synchronized" independent checkpointing because checkpoints with the same checkpoint number on different processors are taken at approximately the same time although they may not be consistent. By taking advantage of this loose synchronization, we implement an algorithm to detect the non-state messages with low overhead. Each processor maintains the following two boolean arrays of size N:

1. *Ever_Recv*[N]: processor p_j sets *Ever_Recv*[i] to 1 when it first receives a message from the corresponding checkpoint interval of p_i. Also, each message sent to p_k is tagged with *Ever_Recv*[k].

2. *Non_State*[N]: processor p_i sets *Non_State*[j] to 1 when it first receives a message from the corresponding checkpoint interval of p_j with *Ever_Recv*[i] equal to 1. If the current checkpoint interval number is l, this means p_i detects an edge from CP_{il} to $CP_{j(l+1)}$. According to Theorem 1, p_i does not have to log any message from the lth checkpoint interval of p_j because all of them are identified as non-state messages.

The four hypercube programs are *QR decomposition*, *Simplex algorithm*, *Cell placement* and *Channel router*. They are executed on an 8-node cube and the execution times are listed in Table 1. The checkpoint interval for each program is arbitrarily chosen to be approximately one tenth of the execution time.

Table 1: Execution and checkpoint parameters of the hypercube programs.

Benchmark programs	QR decomposition	Simplex algorithm	Cell placement	Channel router
Execution time (sec)	385.5	515.5	322.7	442.0
Checkpoint interval (sec)	35	50	35	40
Average rollback distance (CPI)	1.47	0.88	2.10	2.45

The last row of Table 1 shows the *average rollback distance* over five runs for each program in terms of the number of checkpoint intervals (CPI). Here we only consider the global recovery line. Average rollback distances in terms of the local recovery lines should be smaller than the numbers shown. Since each run lasts for approximately ten checkpoint

intervals, the worst-case average rollback distance is $(0 + 10)/2 = 5$ checkpoint intervals which occurs when the domino effect persists during the entire execution and the global recovery line remains at the very beginning of the execution. The average rollback distance for coordinated checkpointing scheme is approximately $(0 + 1)/2 = 0.5$ checkpoint intervals. The actual number should be slightly higher because a certain amount of time has to be spent for receiving and logging the messages comprising the channel states before the most recent recovery line is successfully established [17]. The following remarks are made based on the fact that the average rollback distances shown in Table 1 range from 0.88 to 2.45 checkpoint intervals for the four programs:

1. The domino effect does occur with an independent checkpointing scheme but the average rollback distances are acceptable at least for the four programs tested.

2. Applications with independent checkpointing can benefit from the techniques for reducing rollback propagation (e.g., [24]) and for reducing the space overhead (e.g., the PCSR algorithm).

Table 2 shows the average overhead for message logging. The number of logged messages is only a very small portion (less than 0.5 percent) of the total number of messages. This is also true when message size is considered. Since the checkpoint sizes range from hundreds of kilobytes to several megabytes, even the largest message log per checkpoint which is several kilobytes adds very little extra overhead to the checkpoint. The result shows that Theorem 1 and our algorithm for detecting non-state messages are effective in reducing the overhead for message logging.

Fig. 9 compares our PCSR algorithm with the traditional checkpoint space reclamation algorithm for a typical execution of the Channel router program. Since obsolete checkpoints must be discardable, the curve for non-discardable checkpoints is always below the curve for

Table 2: Message logging overhead in terms of the number and the size of logged messages.

Benchmark programs	QR decomposition	Simplex algorithm	Cell placement	Channel router
Total number of messages	153,551	60,355	254,708	362,513
Number of logged messages	35	17	460	1,345
Percentage	0.02%	0.03%	0.17%	0.37%
Total size of messages (M bytes)	491.7	44.4	13.8	6.7
Size of logged messages (bytes)	112,085	8,432	17,476	23,002
Percentage	0.02%	0.02%	0.13%	0.34%
Average checkpoint size (M bytes)	0.878	3.674	0.690	0.258
Average message log size per CP (bytes)	1,304	105	250	268
Percentage	0.15%	0.003%	0.04%	0.10%

non-obsolete checkpoints. Note that the curves in Fig. 9 do not show the actual number of checkpoints kept on stable storage during program execution because the checkpoint space reclamation algorithm is not continuously active throughout the program execution. Instead, it shows the number of checkpoints which have to be kept if the algorithm is invoked after a certain number of checkpoints have been taken. The figure shows that the PCSR algorithm is particularly effective when the domino effect persists, for example, between the 48th and the 80th checkpoints.

7 CONCLUSIONS

Recording the channel state through message logging allows the checkpointing to be

performed at the application level without the access of low-level communication protocol. Our optimistic logging scheme delays message logging until the time the next checkpoint is taken. The number of messages which have to be logged can be greatly reduced by monitoring the message exchange during the current checkpoint interval. By introducing the idea of valid checkpoints, rollback propagation algorithm developed for schemes without message logging can still apply by simply excluding invalid checkpoints from the graph. A new checkpoint space reclamation algorithm is presented for identifying the set of all discardable checkpoints, which includes the set of obsolete checkpoints as a subset. Communication-trace-driven simulation for four hypercube programs using loosely synchronized independent checkpoir ng scheme shows that our algorithms for reducing checkpointing and message logging overhead are effective.

ACKNOWLEDGEMENT

The authors wish to express their sincere thanks to Pi-Yu Chung for her valuable discussions, to Junsheng Long, Mike Peercy, Craig B. Stunkel and Balkrishna Ramkumar for their help with collecting the communication traces, and to Prith Banerjee for use of his parallel programs.

References

[1] Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," in *Proc. Int. Conf. on Parallel Processing*, pp. 32–41, 1984.

[2] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[3] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. on Software Engineering*, vol. SE-13, pp. 23–31, Jan. 1987.

[4] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 2–11, 1991.

[5] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic recovery schemes for distributed processes," in *Proc. IEEE 2nd Symp. on Reliability in Distributed Software and Database*, pp. 124–130, 1981.

[6] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems*, vol. 3, pp. 204–226, Aug. 1985.

[7] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 3–12, 1988.

[8] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. of Algorithms*, vol. 11, pp. 462–491, 1990.

[9] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault-tolerance," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 90–99, 1983.

[10] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. on Computer Sysiems*, vol. 7, pp. 1–24, Feb. 1989.

[11] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 100–109, 1983.

[12] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," in *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pp. 223–238, 1989.

[13] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. on Computer Systems*, vol. 1, pp. 222–238, Aug. 1983.

[14] D. B. Johnson and W. Zwaenepoel, "Transparent optimistic rollback recovery," *Operating Systems Review*, pp. 99–102, Apr. 1991.

[15] S. F. Nugent, "The iPSC/2 direct-connect communications technology," in *Proc. 3rd ACM Hypercube Conf.*, pp. 384–390, 1988.

[16] P. Pierce, "The NX/2 operating system," in *Proc. 3rd ACM Hypercube Conf.*, pp. 384–390, 1988.

[17] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 12–20, 1991.

[18] B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[19] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs, "Reducing space overhead for independent checkpointing," Tech. Rep. CRHC-92-06, Center for Reliable and High-Performance Computing, University of Illinois at Urbana–Champaign, 1992.

[20] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Comm. of the ACM*, vol. 21, pp. 558–565, July 1978.

[21] K. P. Bogart, *Introductory combinatorics.* Pitman Publishing Inc., Massachusetts, 1983.

[22] I. Anderson, *Combinatorics of finite sets.* Clarendon Press, Oxford, 1987.

[23] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpointing for parallel programs," in *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 79–88, Mar. 1990.

[24] Y. M. Wang and W. K. Fuchs, "Scheduling message processing for reducing rollback propagation." Submitted to *Symp. on Fault Tolerant Computing*, 1992.

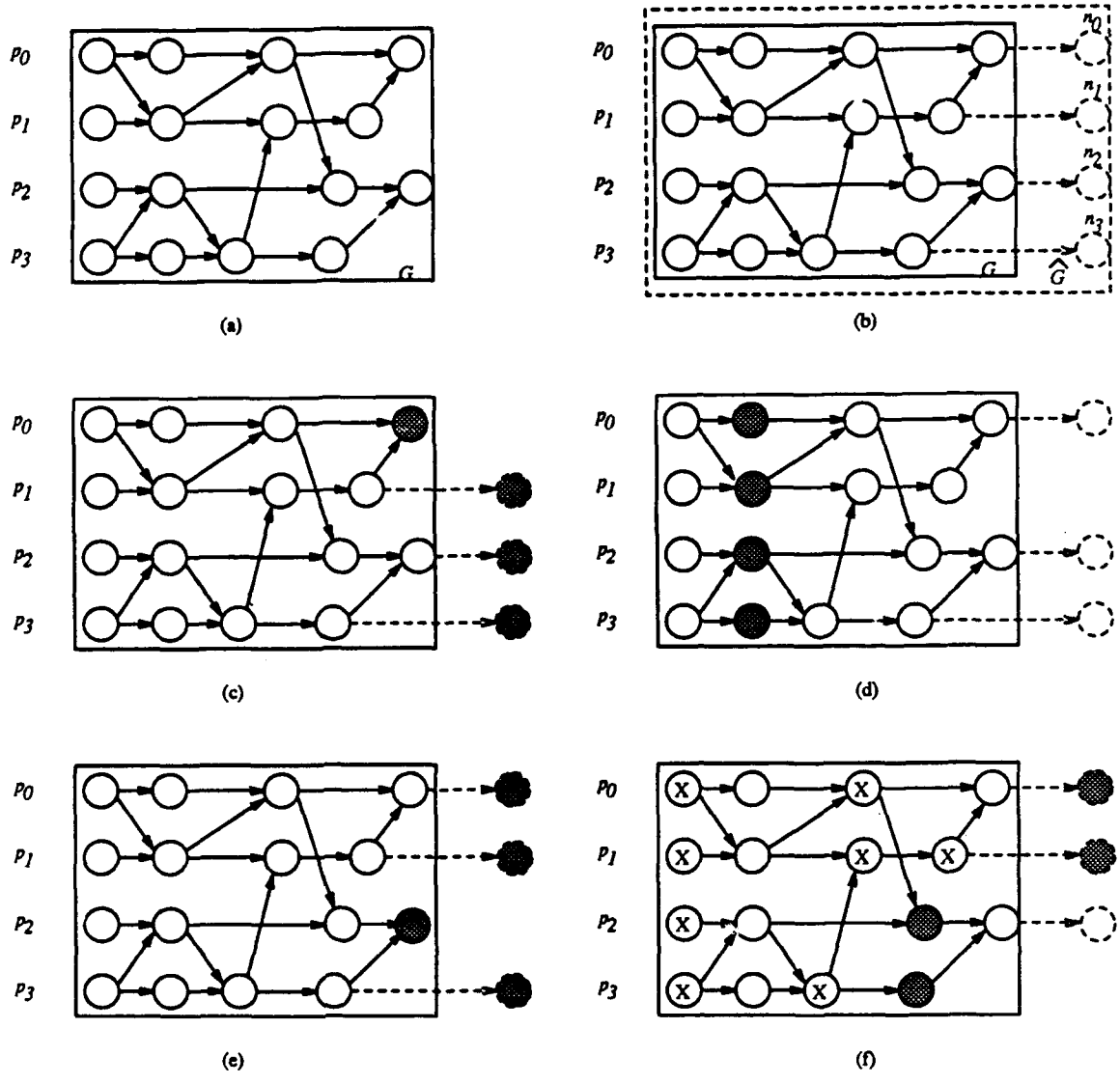

Figure 8: The execution of the PCSR algorithm (a) checkpoint graph G (b) supergraph $\hat{G}$ (c) $\hat{G} - n_0$ (d) $\hat{G} - n_1$ (e) $\hat{G} - n_2$ (f) $\hat{G} - n_3$. (Shaded checkpoints belong to the recovery lines and checkpoints marked "X" are discardable.)

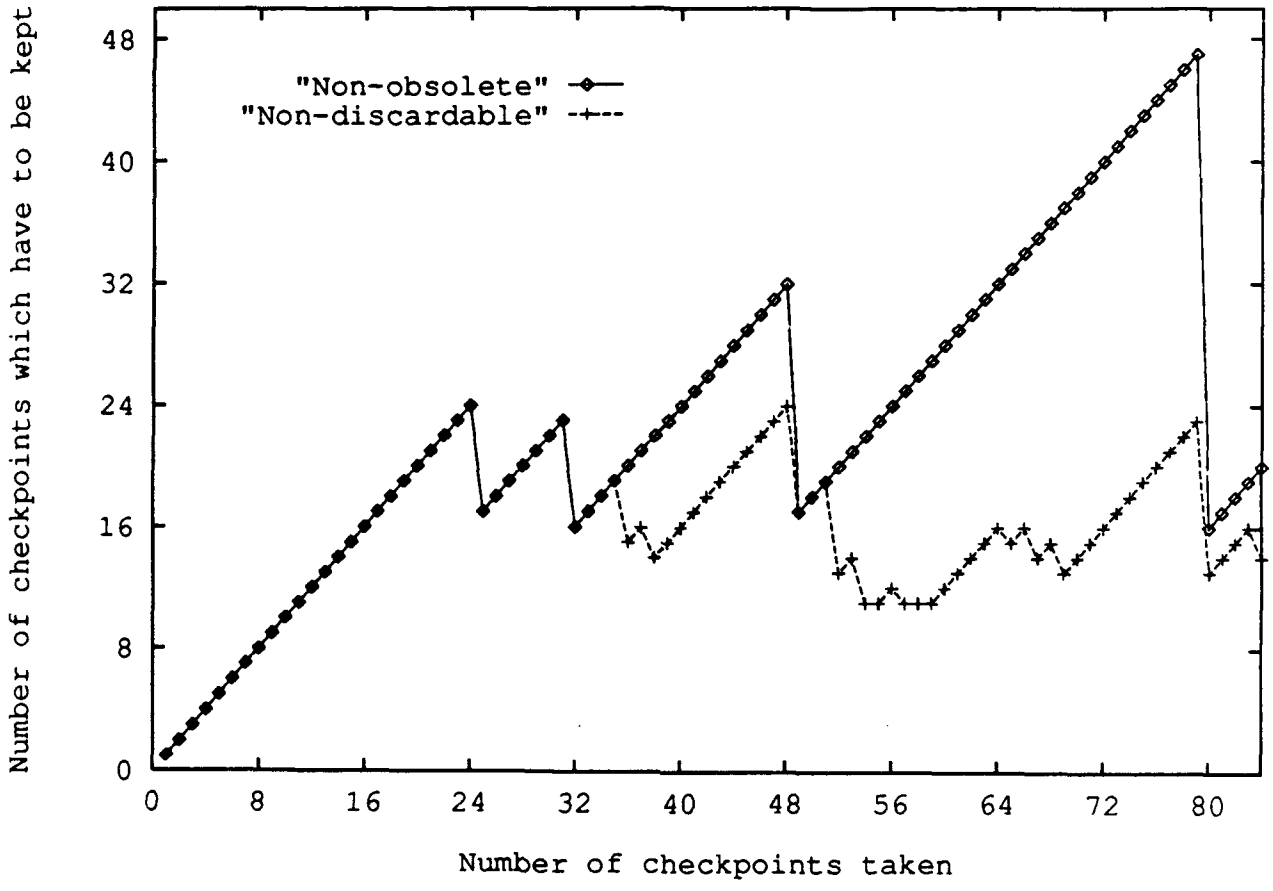

Figure 9: Comparison of the number of non-obsolete checkpoints and non-discardable checkpoints for the Channel router program.